ALL OOPS CONCEPTS USING JAVA WITH 1 TINY PROGRAM – EXPLAINED!

PROGRAM: ENJOY COFFEE IN THE CAR!

Choose your luxury car and enjoy coffee or tea as per your choice while driving.

CREATED BY

Rakesh Singh

A software professional with decades of experience. Author of 2 powerful books for your career growth published on Amazon Worldwide.

1. ***OOP Concepts Booster***
2. ***IT jobs Made Easy For Freshers***

WEBSITE: https://www.interviewsansar.com/

I'll demonstrate and explain all OOP features using this simple application. I'll not focus on designing the application but the simple way, so you can understand clearly.

First, you need to have a look at the complete program, and then come back while reading the concepts given after it, if required.

Table of Contents

COMPLETE PROGRAM IN JAVA

```
import java.util.Scanner;

//Driver class with driver's name and Drive
//functionality
class Driver {
	String name;
	int license;
	int mobile;

	public Driver() {
		this.name = "Car Owner";
		this.license = 11111;
		this.mobile = 11111;
	}

	public void drive() {
		System.out.println("Drive started..." + "Have a
nice drive!");
	}

	public void profile(String name) {
		this.name = name;
	}

	public void profile(String name, int license) {
		this.name = name;
		this.license = license;
	}

	public void profile(String name, int license, int
mobile) {
		this.name = name;
		this.license = license;
		this.mobile = mobile;
	}

	public String getName() {
		return name;
	}
}
```

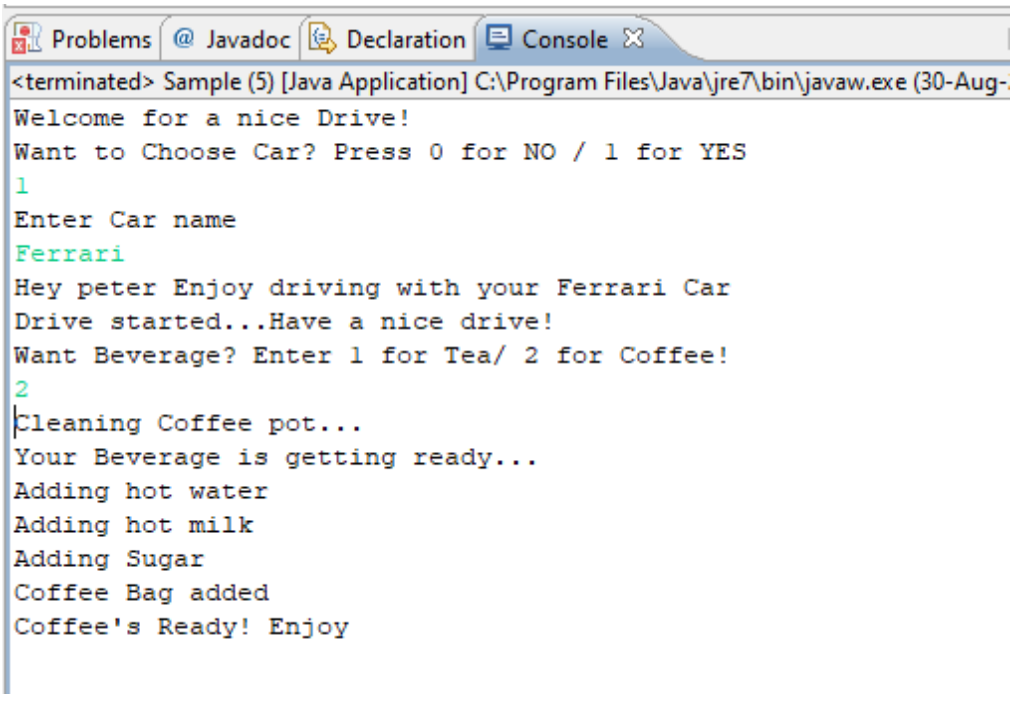
```
// Car class aggregated with driver and
// composed with in built feature beverages
// Tea and Coffee.
class Car {
	Driver driver;
	Beverages b;
	String carChoice;

	Car() {
		this.carChoice = "SUV";
	}

	Car(String carChoice) {
		this.carChoice = carChoice;
	}

	void GetInTheCar(Driver driver) {
		System.out.println("Hey " + driver.getName()
				+ " Enjoy driving with your " +
this.carChoice + " Car");
		driver.drive();
	}

	void EnjobyBeverages() {
		System.out
				.println("Want Beverage?" + " Enter 1
for Tea/ 2 for Coffee!");
		Scanner s = new Scanner(System.in);
		int choice = s.nextInt();
		if (choice == 1) {
			b = new Tea();
		}
		if (choice == 2) {
			b = new Coffee();
		}

		b.getBeverage();
	}
}
```

```
// Beverages abstract class and
// 2 subclasses Tea and Coffee.

abstract class Beverages {

	private void addHotWater() {
		System.out.println("Adding hot water");
	}

	private void addMilk() {
		System.out.println("Adding hot milk");
	}

	private void addSugar() {
		System.out.println("Adding Sugar");
	}

	public void getMixture() {
		System.out.println("Your Beverage is " + "getting
ready...");
		addHotWater();
		addMilk();
		addSugar();
	}

	public abstract void getBeverage();

	public abstract void addIngredients();
}
```

```
// Interface to enforce subclasses to implement
// cleaning tea and coffee pot.

interface Clean {
	void cleanPot();
}

class Tea extends Beverages implements Clean {

	@Override
	public void addIngredients() {
		System.out.println("Tea Bag added");
	}

	@Override
	public void getBeverage() {
		cleanPot();
		getMixture();
		addIngredients();
		System.out.println("Tea's Ready! Enjoy");
	}

	@Override
	public void cleanPot() {
		System.out.println("Cleaning tea pot...");
	}
}

class Coffee extends Beverages implements Clean {

	@Override
	public void addIngredients() {
		System.out.println("Coffee Bag added");
	}

	@Override
	public void getBeverage() {
		cleanPot();
		getMixture();
		addIngredients();
		System.out.println("Coffee's Ready! Enjoy");
	}

	@Override
	public void cleanPot() {
		System.out.println("Cleaning Coffee pot...");

	}
}
```

```
// SAMPLE CLIENT PROGRAM
public class Sample {

	public static void main(String[] args) {
		System.out.println("Welcome for a nice Drive!");
		Scanner s = new Scanner(System.in);
		Driver peter = new Driver();
		peter.profile("peter");
		Car c;

		System.out.println("Want to Choose Car? "
				+ "Press 0 for NO / 1 for YES");
		int carType = s.nextInt();
		if (carType == 1) {
			System.out.println("Enter Car name");
			String carName = s.next();
			c = new Car(carName);
		} else {
			c = new Car();
		}

		c.GetInTheCar(peter);
		c.EnjobyBeverages();
	}

}
```

OUTPUT:



CLASS AND OBJECT

A class is a blueprint or template from which you create multiple **similar** objects. For example, Peter, Jhon, and Linda, etc.

Form code:

Suppose we got a requirement to have a driver profile such as *name*, *license* and *mobile* number, and a functionality ***drive***. Then you can create a template as a Driver class as shown in the program.

If you don't have a template, then an object can try to put his/her age, gender, etc besides the name, license, and mobile, etc. Right?

Objects:

You can see multiple objects created in the code, e.g. for Driver, Tea and Coffee class, etc using the new keyword.

When you create an object using new, it gets created on heap memory. For example,

```
new Car();
c = new Car(carName);
Driver peter = new Driver();
```

NOTE:
1) "new Driver();" statement creates an object on the heap and assigns its reference to the variable peter which is on stack memory.

2)Note that a matching (empty or with parameter) constructor is always called when you create an object of a class.

3) If you don't have a constructor in a class, the default one will be provided by the compiler. If you write a constructor even with a parameter only, the compiler will not provide any constructor. So, you've to write one with an empty parameter.

CLASS CONSTRUCTOR

Constructors have been used in the class ***Car*** and ***Driver*** to initialize objects. In other words, their class fields.

Here are the constructors for the classes Driver and Car and why they're used:

The Driver class constructor is used to initialize with default values. So, if you don't supply value from outside of the class, the default one will be used. Since, I wanted to give an option to set name, license, and mobile for the driver profile, I used a constructor to initialize the fields.

For example, if you only supply your name in the profile, your name will be used in the program. If you don't then the default one will be used.

```
public Driver() {
	this.name = "Car Owner";
	this.license = 11111;
	this.mobile = 11111;
}
```

The Car class contains 2 constructors: 1) with empty parameter, 2) with one String type parameter.

NOTE: Having multiple constructors with different data types or different parameters is known as constructor overloading (compile-time) polymorphism.

From code: the Car class's overloaded constructors are:

```
Car() {
	this.carChoice = "SUV";
}

Car(String carChoice) {
	this.carChoice = carChoice;
}
```

The matching constructors will be AUTOMATICALLY called, when the statements c = **new** Car(); and c = **new** Car(carName); executes in the Sample class.

Why used overloaded constructors in the Car class?

Because a driver has an option to choose a car. If he does not choose a Car and say NO, the default "SUV" car will be automatically selected.

```
System.out.println("Want to Choose Car? "
                  + "Press 0 for NO / 1 for YES");
      int carType = s.nextInt();
      if (carType == 1) {
            System.out.println("Enter Car name");
            String carName = s.next();
            c = new Car(carName);
      } else {
            c = new Car();
      }
```

POLYMORPHISM

"Simply, single functionality with the same name, with different implementation".

1. ***Constructor overloading – compile-time – already explained above.***
2. ***Method overloading – compile-time***
3. ***Method overriding – run-time***

METHOD OVERLOADING

How you overloaded constructors, you can overload **methods with the same name** as well.

In the Driver class, we've 3 overloaded ***profile*** methods, because wanted to give options to users to use any of them. Users may want to set the only name, or name and license, etc.

Why overloaded methods, if you can write different methods with a different name?

Because, as an example, having 10 different method names for the same functionality is difficult to remember, but with the **same name** with different arguments is easier and readable.

From the below examples, you decide which one you'd love to use?

//overloaded methods

```
public void profile(String name)
public void profile(String name, int license)
public void profile(String name, int license, int mobile)

//Methods with different name
public void profileWithName(String name)
public void profileWithNameAndLicense(String name, int license)
public void profileWithNameAndLIcenseAndMobile(String name, int
license, int mobile)
```

NOTE: Method overloading is a compile-time polymorphism, similar to constructor overloading.

METHOD OVERRIDING

You write a method with the same name in subclasses as present in the base class. It can be from base classes: interface, abstract class, a normal.

Form code:

We've overridden and implemented the *cleanPot*() method of interface ***Clean*** into the subclasses Tea and Coffee.

We've overridden *getBeverage*(); and *addIngredients*(); methods of the Beverages abstract class into the Tea and Coffee classes.

Same you can override the method from a normal base class and give it your definition into the child class if you want to use functionalities from the base class but don't want to use some of them.

INHERITANCE

Re-use to functionalities of existing base class to save your time and efforts. And more…avail the feature of inheritance to implement an interface or use and implement the abstract class.

We've inherited and re-used the pre-built functionalities of the Beverages class like *addHotWater(), addMilk(),* and *addSugar(),* etc.

INTERFACE

Simply, provide specifications (contracts) to child classes to implement them.

The **Clean** interface is used to provide clean pot specifications to subclasses Tea & Coffee to implement. In the program, it's used to ENSURE that both the classes Tea and Coffee implement that as a mandatory cleaning process before preparing tea or coffee.

ABSTRACT CLASS

Abstract class acts as a base class and its primary purpose is to have common functionalities of all subclasses at one place in the base class and defer (postpone or force) some functionalities to subclasses to implement them.

In the given program, the Beverages abstract class has common functionalities like add hot water, add milk and add sugar, so the subclasses can inherit and use them. And the abstract class forces subclasses Tea and Coffee to implement getBeverage() and addIngredients() functionalities.

ABSTRACTION & ENCAPSULATION

Abstraction means providing only essentials to the users, and Encapsulation means hiding the complexities in **whatever way** you can hide.

Actually, abstraction is design-level concept where you decide what is necessary information to provide to users, and encapsulation is implementation level.

Form the code:

In the abstract class Beverages, we could have provided the functionalities *addHotWater*(), *addMilk*(), and *addSugar*() to the **users,** Tea, and Coffee subclasses.

But, I thought of abstraction, why give more responsibilities to users Tea and Coffee classes to call 3 functions? Can't we reduce their responsibilities by providing only one getMixture() interface to them?

The abstraction says here: Only getMixture() method to users are enough.

So, I applied encapsulation at the implementation level design. Made all the 3 methods private and implemented one public method getMixture(). In this public method, I wrapped all the 3 methods.

COMPOSITION AND AGGREGATION

These are the relationship among the objects. In composition, the main object is destroyed, all the composed objects also MUST be destroyed whereas, in Aggregation, the composed object must not be destroyed.

These are the concepts and totally up to you how you want to maintain relationships between objects primarily to handle memory uses occupied by the objects.

From the code:

In the program, the Car class composes the objects of Tea and Coffee. Give special attention that the object of Tea and Coffee is created using the new keyword inside the Car class in a method EnjobyBeverages(). So, when the Car class object is destroyed, immediately the Tea and coffee class object MUST also be destroyed.

In the Car class, the Driver is aggregated, its object is created using new outside of the class in the Sample class. So, when the car is destroyed, the Driver SHOULD NOT be destroyed.

GENERALIZATION AND SPECIALIZATION

Generalization means you move all the common functionalities of the child classed to a base class to avoid duplicate code and make code cleaner and readable.

All the child classes Tea and Coffee could have the methods add hot water, add milk and sugar. But they are generalized and move to the abstract class Beverages.

Specialization: The specialized functionalities are move to the child classes to implement their own way. For example, the *getBeverage*() and *addIngredients*() functionalities are given to the Tea and Coffee child classes of the Beverages class to implement their own way.

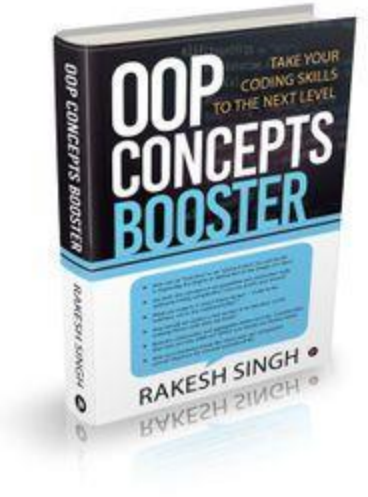
Hope you Enjoyed Reading!

2 NEW BOOKS RECOMMENDATAION FOR YOUR CAREER GROWTH

AVAILABLE @ AMAZON Worldwide & Flipkart

1- OOP Concepts Booster

You can make your OOPs Concepts rock solid with the book OOP Concepts booster, so you can stand out from the crowd in Job interviews or companies or college.



This book will teach you best utilization, purpose and uses of powerful oop components with real time examples besides clarifying the oop concepts doubts that you don't find in tutorials book or internet.

Here's how it helps you Master:

This contains **fine blended notes with important points on oops topics**, and **25 Q&A with practical** Real-Time Examples to enhance your coding skills up to industry-level code.

Here's the book content:

Quick Notes:

1. Class & Object, 2.Constructor, 3. Inheritance, 4. Polymorphism, 5. Interface,6. Abstract Class & Abstract Method, 7. Abstraction & Encapsulation, 8. Singleton Class.

OOP Questions:

Q-1) What are the memory view of the objects and the references of a class? When is the memory allocated to them cleared and who clears this memory? What is the lifespan of the objects and their references?

Q-2) What is the effect of a private constructor of a class and in what scenarios can it be beneficial?

Q-3) Why are method overloading and method overriding called compile-time and run-time polymorphism respectively? What can be the code example scenarios to illustrate the compile-time and run-time activities?

Q-4) What are the scenarios where a static method is mandatory?

Q-5) What can be the issue if you delete a base class method if a subclass overrides it?

Q-6) Why use the interface reference for subclass objects while the subclass reference works as well?

Q-7) How do polymorphism and inheritance provide extensibility?

Q-8) Why should anyone use constructor overloading? How does this help?

Q-9) How does inheritance help eliminate duplicate code?

Q-10) Why do you need to overload the method if methods with different names do the task as well?

Q-11) What are the multiple ways to reuse the code in OOP?

Q-12) If we can't create an object of an abstract class, then what is its purpose? What can be the scenarios of uses?

Q-13) How does encapsulation provide security?

Q-14) Are both the concepts of encapsulation and abstraction really related to hiding complexities? Can you Justify your answer?

Q-15) How should we update a new version of an interface, so the existing client's code does not break?

Q-16) Which one is a good choice if you have an option to choose between an interface and an abstract class and why?

Q-17) What are the main aims of using an interface? Do we really use interface variables? If so, for what?

Q-18) What are impacts if I don't follow dictum – "Code to the interface, not to the implementation"?

Q-19) How can an "interface" or an "abstract class", be used to aid in improving the degree of abstraction in the design of a class?

Q-20) Both the composition and aggregation follow the Has-A relationship, then how are they different? Which one should you choose when?

Q-21) In what scenarios inheritance is indispensable? Is it true that inheritance is used for code reuse only? If not, what are the other factors?

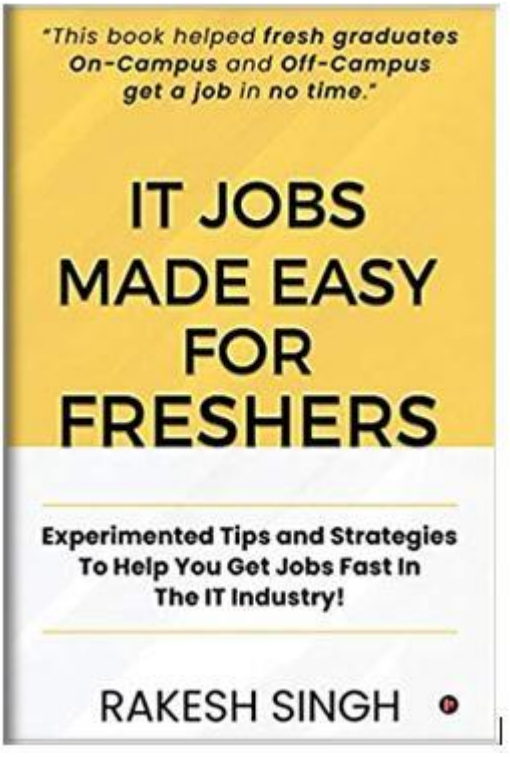
Q-22) Is really multi-level inheritance used? What can be an example of it in real-time?

Q-23) How should an interface be designed so that no client is compelled to depend on specifications that it does not use?

Q-24) How can inheritance break the client code, but composition cannot? Illustrate the scenario example of this.

Q-25) What are the good guidelines to choose inheritance or composition?

2 - IT JOBS MADE EASY FOR FRESHERS: Experimented Tips & Strategies to help you get IT jobs.



This book I wrote for freshers to share my working guidelines to get a software job.

CSE Engg. students can read and prepare from their 3rd year.

For long years, I experimented with my tips and strategies being with fresh graduates, and all got job in 5 months. – ALL OF THEM…And that's from nearly ZERO preparation.

In these competitive world, it seems almost impossible to get a job as a freshers, but its not… if you prepare and search for job in the right way. Which is difficult to know at the fresher level.

Getting a job is easy if you know what recruiters, organizations, and job portals are looking for. When you know…

- The simple activities, and writing your resume in such a way that engage recruiters out of other hundreds of resumes, excite them, and makes them call you for an interview. – writing objectives, address, project description, photo, styling etc… simply DOESN'T work.
- What skills are enough and up to what level to get a job, so you can finish in less time, and the tactics to handle the written test.
- How to answer correctly to impress any interviewer
- How to make yourself stand out from the crowd with simple and doable activities, so you can land your job faster and easier.

…And so much more! **In fact, 50+ practical tips – All is packed in your life-changing book** "**IT jobs made easy for freshers**", **to get hired in record time,** Published in 2020.

Few amazons book review is available because it's a new book.

You can start reading the book if you're a fresher to get a job faster.